Hindawi
Scientific Programming
Volume 2017, Article ID 3409647, 12 pages
https://doi.org/10.1155/2017/3409647

Research Article

MPI to Coarray Fortran: Experiences with a CFD Solver for Unstructured Meshes

Anuj Sharma and Irene Moulitsas

School of Aerospace, Transport and Manufacturing (SATM), Cranfield University, Cranfield, Bedfordshire MK43 0AL, UK

Correspondence should be addressed to Anuj Sharma; a.sharma@cranfield.ac.uk

Received 5 March 2017; Accepted 20 June 2017; Published 28 September 2017

Academic Editor: Can Özturan

High-resolution numerical methods and unstructured meshes are required in many applications of Computational Fluid Dynamics (CFD). These methods are quite computationally expensive and hence benefit from being parallelized. Message Passing Interface (MPI) has been utilized traditionally as a parallelization strategy. However, the inherent complexity of MPI contributes further to the existing complexity of the CFD scientific codes. The Partitioned Global Address Space (PGAS) parallelization paradigm was introduced in an attempt to improve the clarity of the parallel implementation. We present our experiences of converting an unstructured high-resolution compressible Navier-Stokes CFD solver from MPI to PGAS Coarray Fortran. We present the challenges, methodology, and performance measurements of our approach using Coarray Fortran. With the Cray compiler, we observe Coarray Fortran as a viable alternative to MPI. We are hopeful that Intel and open-source implementations could be utilized in the future.

1. Introduction

1.1. Motivation. While it is the dominant communication paradigm, Message Passing Interface (MPI) has received its share of criticism in the High-Performance Computing (HPC) community. It provides a complex interface to parallel programming, which is mostly underutilised by researchers whose primary skill is not software development. Maintenance and modernization of parallel codes written with MPI also require more person-hours and associated funding costs compared to the serial counterpart [1].

In parallel programming, details of communication strategies should not overbear the researchers, to avoid shifting their focus from the core research objective. Unfortunately, it has been observed that hardware advancements do not come hand in hand with better performances. Scientific codes utilizing the MPI paradigm have to be modified in order to achieve the best possible performance gains. With the goal of Exascale computing, both the underlying hardware and the software tools available should support the scientific numerical codes so that they are efficiently adaptable to future computing platforms. Discussing the efficiency of a scientific code is a twofold matter and it should involve both the effort put during the development or reengineering phase, as well as the performance gains observed later.

Recently, Partitioned Global Address Space (PGAS) based parallel programming languages have been gaining popularity. Several languages such as Unified Parallel C (UPC), Coarray Fortran, Fortress, Chapel, and X10 are based on the PGAS paradigm. In comparison to many of its competitors, Coarray Fortran is relatively mature and has undergone considerable research [2]. It provides a natural syntax to Fortran programmers and generates a lucid code.

1.2. Related Work. Coarray Fortran was originally a small syntactic extension (F−) to the Fortran programming language, which enabled parallel programming. It is now part of the Fortran programming language since the adoption of the Fortran 2008 standards. Some features, such as collective intrinsic routines, teams, and error handling of failed images, were left out in Fortran 2008 standards. With the acceptance of the technical specification document, they will become standard in Fortran 2015 [3].

Like other PGAS languages, Coarray Fortran provides language constructs equivalent to one-sided communication during run-time. This feature improves productivity and could also harness the communication features of the underlying hardware. Some studies have been performed to quantify the effort and performance of such PGAS languages, most notably in the PRACE-PP (Partnership for Advanced Computing in Europe-Preparatory Phase) project. It involved the development of three benchmark cases by different researchers and collecting the feedback of development time (effort) and performance [2]. While Chapel and X10 were found to be immature, UPC and Coarray Fortran were recommended due to their performance, low development time, and relative maturity.

Coarray Fortran is today supported by Cray with extended features and by Intel with compatibility with Fortran standards [4]. Open-source compilers are also in different development stages, such as the GCC compiler (OpenCoarrays [5]) and OpenUH [6].

Over the years many benchmark studies have been performed [7–12] to investigate the performance of Coarray Fortran in comparison to MPI. In [7] Numrich et al. suggest MPI has high bandwidth, but high latency for messages. In contrast, Coarray Fortran has low bandwidth and low latency for messages. This behavior is dependent upon support for remote direct access, by the underlying hardware architecture. The overall performance comparison of MPI and Coarray Fortran is murky with contradicting results, such as, for example, by [7, 8]. The contradiction in the better performance could be attributed to the different communication requirements of various scientific codes. Other studies have focused on individual aspects of parallel programming, that is, memory layout [9], use of derived data types [9], buffered/unbuffered data transfer [10], object-oriented programming [13], and collective communication constructs [11]. One-sided communication with Coarray Fortran has also shown promising result for heterogeneous load balancing on the Intel Xeon Phi architecture [14].

1.3. Objective. Computational Fluid Dynamics (CFD) studies of complex flows in a wide range of applications certainly benefit from parallelization due to the high computational costs of the numerical methods employed. Recent performance studies in the literature have only focused on numerical codes with structured meshes. These codes have natural, geometry driven, grid partitioning, and regular communication patterns. In many scientific domains, where complex geometries are involved, unstructured meshes are the norm. These meshes lead to nonintuitive mesh partitionings, have greater load imbalances, and suffer from nonregular communication patterns. When higher-order numerical schemes are required, the complexity of the communication patterns and associated data structures increases even more. In our study, we present our experience of converting a scientific numerical CFD code with unstructured meshes and higher-order numerical schemes from MPI to Coarray Fortran for parallel communication.

```
Real :: i(5), j(5)[*], k(5)[4,*], l[*]
real, codimension[*] :: m
```

Listing 1: Coarray declaration.

2. Overview

2.1. Coarray Fortran. Coarray Fortran is based on the Single Program Multiple Data (SPMD) model of parallel programming [15, 16]. A set of independent instances of the program, called *images*, executes simultaneously on different processors. The number of *images* can be chosen at compile or run-time and has a unique index (1 to number-of-processors). Fortran standards provide two intrinsic functions - `this_image()` and `num_images()` to retrieve the *image* index and the total number of *images*, respectively.

2.1.1. Coarrays. A *coarray* is similar to an array in Fortran, that is, a collection of data objects, with an exception that it can be accessed by other *images* as well. In comparison, a regular array is private to the parent *image*. *Coarrays* are declared using an additional trailing subscript in square brackets, [], referred as *codimensions*. A *coarray* has *corank*, *coshape*, and *cobounds* similar to corresponding terms for an array. Intrinsic functions to find lower and upper cobounds are `lcobound`, `ucobound`. Examples for different types of valid *coarray* declaration are shown in Listing 1. While `i` is not a *coarray*; `j`, `k`, `l`, and `m` are *coarrays*. `j` and `k` are *coarray* of an array (with 5 elements). `l` and `m` are scalar *coarrays*. The upper *cobound* of the last *codimension* for a *coarray* is always defined as `*`, whose value is dependent upon the number of *images* specified during execution. In Listing 1, the upper *cobound* of `j`, `l`, and `m` would be equal to number of images, while the upper *cobound* of `k` would be equal to `num_image()/4`.

2.1.2. Allocatable Coarrays. An allocatable *coarray* could be used to define the *codimensions* at run-time (see Listing 2). *Cobounds* must be specified, and upper *cobound* should be `*`. Deallocate statement is similar to that of an allocatable array in Fortran language. To declare an allocatable *coarray*, the code in Listing 2 can be used.

Also, the same allocate statement should be executed by all *images* (same *bounds* and *cobounds*); thus *coarrays* cannot have different sizes for different *images*. This limitation is not significant if the data arrays that are used for communication are of equal length on all the images, such as in structured mesh applications. With unstructured meshes and especially with higher-order numerical schemes such as WENO (Weighted Essentially Nonoscillatory), the length of communication array varies widely. To overcome this limitation, derived data types could be used as shown in Listing 3.

2.1.3. Communication: Push versus Pull. Data is remotely accessed using *codimensions* without the conventional *send*

```
real, allocatable :: o[:,:]
...
allocate(o) !Not allowed - coubounds should be specified
allocate(o[2,3]) ! Not allowed - upper cobound should be *
allocate(o[2,*]) ! Allowed
...
deallocate(o)
```

Listing 2: Coarray declaration of an allocatable coarray.

```
type CoData
  integer :: myrank
  real, allocatable :: sol(:)
end type
...
type (CoData), allocatable :: Image[:]
...
allocate(Image[*]) !Allocate derived data type for all
   images
...
allocate( Image%sol(storeSize) ) !storeSize could have
   different values on all images
...
deallocate(Image)
```

Listing 3: Coarray declaration of an allocatable, derived data object coarray.

and *receive* messages used in MPI. To copy data from another *image* using a *coarray*, either *pull* or *push* approach can be used. In the *pull* approach, data is received from another *image*. That is, to copy from the next *image*, one could use Listing 4.

Note that a *coarray* reference without [] indicates a reference to the variable in the current *image*. Similarly, to *push* some data to the next *image*, one may use Listing 5.

Similarly, for an allocatable, derived data type *coarray*, one may use the Listing 6 to *push* data to the next *image*. Usually, the choice between push and pull approach is based on the algorithm used.

2.1.4. Synchronization. As the *images* run asynchronously, care must be taken to maintain correct execution order by specifying explicit synchronization statements. All the participant *images* must execute this statement before any *image* can proceed forward. Synchronization statements are `sync all` (to synchronize all *images*) and `sync images` (for selective synchronization). Implicit synchronization takes place during allocation and deallocation of allocatable *coarrays*.

2.2. CFD Solver. Our CFD solver is an unstructured mesh, finite volume Navier-Stokes solver for compressible flows, supporting mixed element meshes. In certain situations, the compressible nature of the fluid results in shock waves, with a sharp interface between regions of distinct properties such as density and pressure. These flows are commonly encountered in aerospace applications. To avoid prediction of a diffused interface and to predict the shock strength accurately using a CFD solver, higher-order numerical schemes are essential [17].

In a cell-centered finite volume solver, such as ours, the cell volume averaged solution (with either conserved or characteristic variable) is stored at the center of the cells in the mesh. If these cell-averaged values are used for the intercell flux calculations in the iterative solver to determine the solution at next time step or iteration, then first-order spatial accuracy is achieved. For greater accuracy, conservative and higher-order reconstruction polynomial is used. The neighboring cells which are used for calculating the reconstruction polynomial define the zone of influence and are collectively known as the stencil. The order of accuracy of the reconstruction is dependent upon the size of the stencil, while the reconstruction provides greater accuracy in the regions with smooth solutions; near sharp discontinuities such polynomials are inherently oscillatory [18–20].

In the traditional Total Variation Diminishing (TVD) schemes, the oscillatory nature of the polynomial near a discontinuity is kept under control by using slope or flux limiters. Thus, resulting schemes, such as MUSCL scheme (Monotonic Upstream-Centered Scheme for Conservation Laws), have higher-order accuracy in the region with smooth

```
if( this_image().ne. num_images() ) then
  j(:) = j(:)[this_image()+1]
end if
```

Listing 4: Coarray communication: pull approach.

```
if( this_image().ne. num_images() ) then
  j(:)[this_image()+1] = j(:)
end if
```

Listing 5: Coarray communication: push approach.

solutions while accuracy is lowered in regions with sharp or discontinuous solution.

The WENO scheme aims to provide higher-order accuracy throughout the domain by using multiple reconstruction polynomials with solution adaptive nonlinear weighting. The WENO scheme uses one central stencil and several directional stencils to construct the reconstruction polynomial. Higher weighting is given to smoother reconstruction polynomial among the directional stencils, and the highest weighting is given to the central stencil. The nonlinear weights are thus solution adaptive.

Details of the implementation of the CFD solver are provided in [19]. The MPI version of the solver has been used in previous studies for solving Euler equations [18] and compressible Navier-Stokes equations [19].

3. Conversion to Coarray Fortran

The MPI version of the code uses different derived data types to store the values of the solution variables and the associated mesh data. Since the code uses unstructured meshes and the WENO scheme, it has inherent load imbalance due to stencils of varying lengths. To accommodate the imbalanced memory storage, derived data type *coarrays* with allocatable components are essential; according to the Fortran 2008 standard, *coarrays* of standard data types must have the same size on all the *images*.

For simplicity, a generic naming scheme in the following text to explain the modifications required in the code to incorporate communication with Coarray Fortran.

Let us say, in an *image* or a process, the child data (i.e., the allocatable array) which should be sent is `SendData` and the parent data (i.e., the allocatable array of derived data type) holding many such `SendData` is `SendArray`. Similarly, let the array names for the receiving child data and parent data type be `ReceiveData` and `ReceiveArray`.

In the MPI version, every process has its `SendArray` and `ReceiveArray` which also hold the process numbers to which the data is to be sent, and from which process the data is to be received. The memory location at which data is to be sent is not stored in the sending process. In the combined send receive MPI subroutine, `MPI_SendRecv`, a process sends data as a message which is received by another process, which decides where the data is to be saved (see Figure 1).

In the MPI version, respective `SendData` is sent to all the receivers among all the processes. When this data is received, the received data is stored in `ReceiveData` by all the receiving processes.

3.1. Construction of Communication Array. To incorporate the Coarray Fortran communication, with minimal changes to the original data structure and to avoid any additional memory copies before and after communication, an additional communication array was created.

Since push communication was needed in the Coarray Fortran version as well, the location of the `ReceiveData` in `ReceiveArray` should be known to the sender. To store this location, an additional array was created, referenced in the code as `CommArr *`, where $*$ denotes a number specified to set different variable apart. For this discussion, let us give it a generic name, `CommArr`, and call it as communication array.

An initialization subroutine is called once before the communication subroutine to find the `ReceiveData` location for a receiving image. This information is stored in every sender image. Communication with Coarray Fortran becomes simpler once the `CommArr` is set up. A sender image now directly transfers data to a receiver image's memory using Coarray Fortran syntax (see Figure 2).

3.2. Working of the Communication Array. `CommArr` provides connectivity between the `SendArray` of the sender image and `ReceiveArray` of the receiving images. Along with other data, `SendArray` also stores the index of the receiving image. Thus, it also serves as an input for the `CommArr` array, which stores the position at which receive image has allocated memory for receiving the data. Since Fortran 2008 standard requires same coarray bounds, `CommArr` has an upper bound equal to the number of processes, and empty values in `CommArr` are filled with -1.

Figure 3 explains the working of communication array using two examples shown by solid and dashed lines. In the first example, *Image* 1 needs to send data to *Images* 2, 3, and

```
if( this_image().ne. num_images() ) then
  Image[this_image()+1]%sol(:) = Image%sol(:)
end if
```

Listing 6: Coarray communication: push approach with a derived data type coarray.

4. While sending data to *Image* 3, *Image* 1 uses `CommArr`. At location 3, `CommArr` of *Image* 1 stores 1. This is the location at which data need to be sent to the receiving *Image* 3. In the second example, *Image* 3 sends data to *Image* 4. `CommArr` of *Image* 3 store 2 at position 4. 2 is the position at which the receiving *Image* 4 will receive the data.

4. Tests

4.1. Physical Case. A 2D, external flow, test case was chosen for validation and performance measurements. In this test case, air flow over RAE2822 aerofoil in steady, turbulent conditions was modeled in the transonic regime. The computational domain boundaries were fixed 300 chord lengths away, and an unstructured mixed mesh was created which contained quadrilaterals in the boundary layer near the aerofoil and triangular element away from it. The resultant mesh had 52378 cells, 39120 quadrilateral cells, and 13258 triangular cells. The free-stream conditions at the inlet correspond to *Case 6* in the experimental results from literature [21].

$$\begin{aligned} \mathrm{Re} &= 6.5 \times 10^6; \\ M &= 0.725; \\ \alpha &= 2.92. \end{aligned} \tag{1}$$

Here, Re is the Reynolds number, M is the Mach number, and α is the angle of attack. The free-stream conditions in the original literature [21] are not corrected for the wind tunnel effects. Including the wind tunnel effect results in the following free-stream conditions [22].

$$\begin{aligned} \mathrm{Re} &= 6.5 \times 10^6; \\ M &= 0.729; \\ \alpha &= 2.31. \end{aligned} \tag{2}$$

Subsonic boundary conditions were employed on the outer domain, while no-slip boundary conditions were employed on the aerofoil.

The third-order WENO scheme, denoted as WENO-3, was used for achieving higher-order accuracy. For the WENO-3 scheme, the central and the directional stencils for a triangular mesh element are shown in Figure 4. The zone of influence from the four stencils is shown in Figure 5. This zone of influence is considerably larger than the traditional second-order numerical schemes, which contributes toward the computational and communication costs.

4.2. Hardware and Compiler. Two HPC facilities were used in our study, ASTRAL and ARCHER. ASTRAL is an SGI, Intel processor based, cluster owned by Cranfield University. ARCHER, the UK's national supercomputing facility, is a Cray XC30 system.

ASTRAL has 80 physical compute nodes. Each compute node has two 8-core E5-2260 series processor and 8 GB RAM per core (i.e., 128 GB per node). Hyperthreading is disabled. A 34 TB parallel file storage system (Panasas) is connected to all the nodes. Infiniband QDR connectivity exists among all nodes and to the storage appliance. The operating system is Suse Linux 11.2.

ARCHER has a total of 4920 compute nodes. Each standard compute node (4544 nodes out of the total 4920 nodes) contains two 12-core E5-2697 v2 (Ivy Bridge) series processors and a total of 64 GB RAM per node. Each processor can support two hyperthreads, but they were not used. The compute nodes are connected with a parallel Lustre filesystem. The Cray Aries interconnect links all the compute nodes. A stripped-down version of the CLE, Compute Node Linux (CNL), is run on the compute nodes to reduce the memory footprint and overheads of the full OS.

Intel Fortran compiler 15.0.3 and Intel MPI version 5.0 Update 3 were used on ASTRAL and Cray compiling environment 8.3.3 was used on ARCHER.

The compiler flags used on ASTRAL were `-i4 -r8 -O2 -fp-model source`.

The compiler flags used on ARCHER were `-s interger32 -s real64 -e0 -ea -eQ -ez -hzero -eh -eZ`.

5. Results

5.1. Clarity. Coarray Fortran uses a simpler and user-friendly syntax, which results in a cleaner code in comparison to MPI.

To demonstrate the clarity obtained with the Coarray Fortran, we have presented one communication subroutine from our code in Listing 7 (MPI) and Listing 8 (Coarray Fortran). For simplicity, only the source code for the communication is shown.

This subroutine is used for communicating the reconstructed, boundary extrapolated values of each Gaussian quadrature point of the neighboring halo cells. The derived data type (`iexboundhir` and `iexboundhis`) used for the communication has the following declaration shown in Listing 9.

The resultant code is also much easier to understand, while preserving the functionality.

```
subroutine exhboundhigher( n, iexchanger, iexchanges,
 iexboundhir, iexboundhis, itestcase, numberofpoints2,
 isize )
...
  if (itestcase .eq. 4) then
    do i=0, isize-1
      if (i .ne. n) then
        do k=1, indl
          if (iexboundhir(k)%procid .eq. i) then
            do j=1, tndl
              if (iexboundhir(k)%procid .eq. iexboundhis(j)
                  %procid) then
                call mpi_sendrecv( iexboundhis(j)%facesol
                   (1:iexchanges(j)%muchtheyneed(1), 1:1,
                   1:numberofpoints2, 1:5), &
                  iexchanges(j)%muchtheyneed(1)*1*
                      numberofpoints2*5,
                      mpi_double_precision, iexboundhis(j)%
                      procid, n, &
                  iexboundhir(k)%facesol(1:iexchanger(k)%
                      muchineed(1), 1:1, 1:numberofpoints2,
                      1:5), &
                  iexchanger(k)%muchineed(1)*1*
                      numberofpoints2*5,
                      mpi_double_precision, iexboundhir(k)%
                      procid, &
                  iexboundhir(k)%procid, icommunicator,
                      status, ierror )
                call mpi_sendrecv( iexboundhis(j)%facesolv
                   (1:iexchanges(j)%muchtheyneed(1), 1:1,
                   1:numberofpoints2, 1:8), &
                  iexchanges(j)%muchtheyneed(1)*1*
                      numberofpoints2*8,
                      mpi_double_precision, iexboundhis(j)%
                      procid, n, &
                  iexboundhir(k)%facesolv(1:iexchanger(k)%
                      muchineed(1), 1:1, 1:numberofpoints2,
                      1:8), &
                  iexchanger(k)%muchineed(1)*1*
                      numberofpoints2*8,
                      mpi_double_precision, iexboundhir(k)%
                      procid, &
                  iexboundhir(k)%procid, icommunicator,
                      status, ierror )
              end if
            end do
          end if
        end do
      end if
    end do
  end if
...
end subroutine exhboundhigher
```

Listing 7: `exhboundhigher` subroutine from the MPI version of the code. Note that only the code responsible for communication is shown.

```
subroutine exhboundhigher ( n, iexchanger, iexchanges,
   iexboundhir, iexboundhis, itestcase, numberofpoints2,
   isize )
  ...
  if (itestcase .eq. 4) then
    do i=1, tndl
      ! add +1 since coarray images are mpi ranks+1
      j = iexchanges(i)%procid + 1
      iexboundhir(CommArr1(j))[j]%facesol(:,:,:,:) =
        iexboundhis(i)%facesol(:,:,:,:)
    end do
    do i=1, tndl
      ! add +1 since coarray images are mpi ranks+1
      j = iexchanges(i)%procid + 1
      iexboundhir(CommArr1(j))[j]%facesolv(:,:,:,:) =
        iexboundhis(i)%facesolv(:,:,:,:)
    end do
  end if
  sync all
  ...
end subroutine exhboundhigher
```

Listing 8: `exhboundhigher` subroutine from the Coarray Fortran version of the code. Note that only the code responsible for communication is shown.

```
type exchange_boundhi
  integer :: procid, fast
  integer :: howmany
  real, allocatable, dimension(:,:,:,:) :: facesol, quadp,
      vert
  real, allocatable, dimension(:,:,:) :: wquad, angles
  real, allocatable, dimension(:,:) :: triap, normals
  real, allocatable, dimension(:,:,:,:) :: facesolv
end type exchange_boundhi
```

Listing 9: Derived data type of iexboundhir and iexboundhis variables in the code.

5.2. Validation. To validate the numerical predictions from the CFD code the experimental measurements [21] and CFD predictions [22] from the literature were used. This comparison was made for both the version of the code, with MPI communication and with Coarray Fortran communication.

The pressure coefficient profile over the aerofoil for the WENO-3 scheme along with the reference results is shown in Figure 6. The pressure coefficient is negative on the top surface of the aerofoil. It corresponds to the pressure less than the free-stream pressure. Pressure coefficient was calculated using

$$C_p = \frac{p - p_\infty}{(1/2)\,\rho_\infty V_\infty^2}, \tag{3}$$

where C_p is the pressure coefficient, p is pressure, ρ is the fluid density, V is the velocity, and subscript ∞ represents free-stream values.

The sharp dip in pressure coefficient, near $x/c = 0.55$ (x is the distance over the aerofoil and c is the chord), represents the shockwave (i.e., sudden change in the value of pressure and density). It can also be seen in Figure 7, which shows the Mach number contours over the aerofoil obtained with the WENO-3 scheme. Mach number 1 represents a shock wave.

It can be observed that the predictions with the WENO-3 scheme are more accurate in predicting the shock location, compared to the WIND code. The WIND code uses second-order finite difference scheme; thus greater errors may be expected. The MPI and Coarray Fortran version of our code provides the same predictions, which reassures that errors were not introduced during the conversion process.

5.3. Performance. To compare the performance of MPI and Coarray Fortran communication in the code, the validation test case was run for 1000 iterations. These tests were performed on ASTRAL (Intel compiler) and ARCHER (Cray compiler), and the elapsed time was measured for the iterative calculations excluding any initialization and savefile outputs.

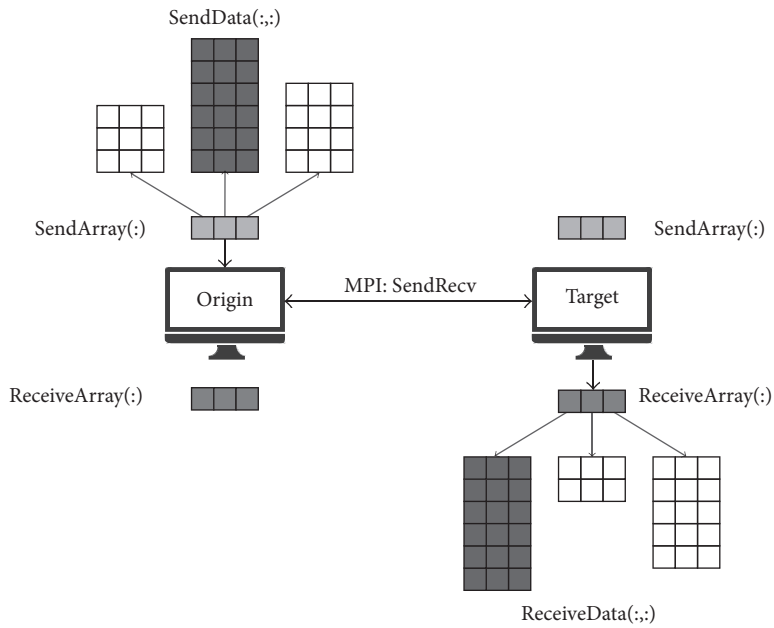

Figure 1: Schematic of data exchange process using MPI.

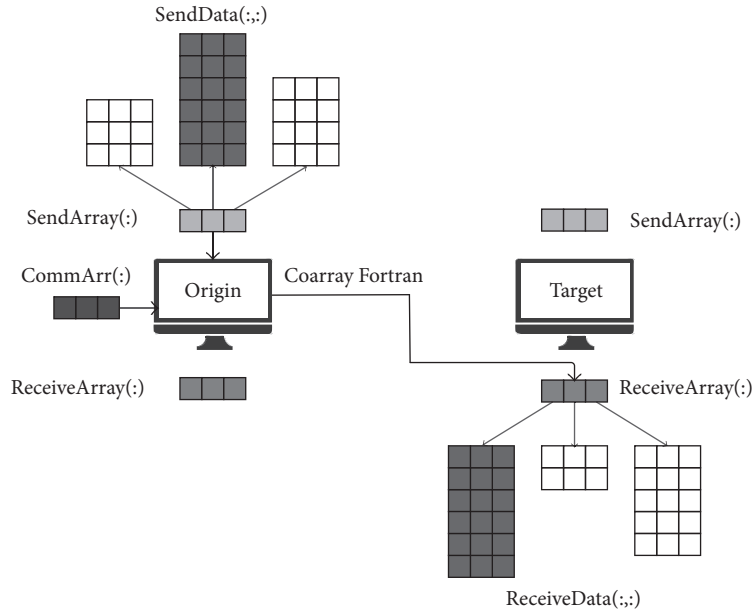

Figure 2: Schematic of data exchange process using Coarray Fortran.

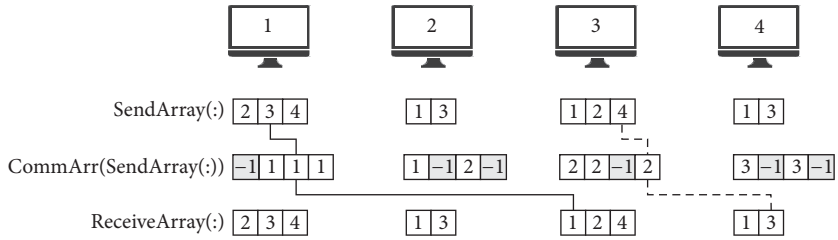

Figure 3: Working of CommArr in the Coarray Fortran version of the code.

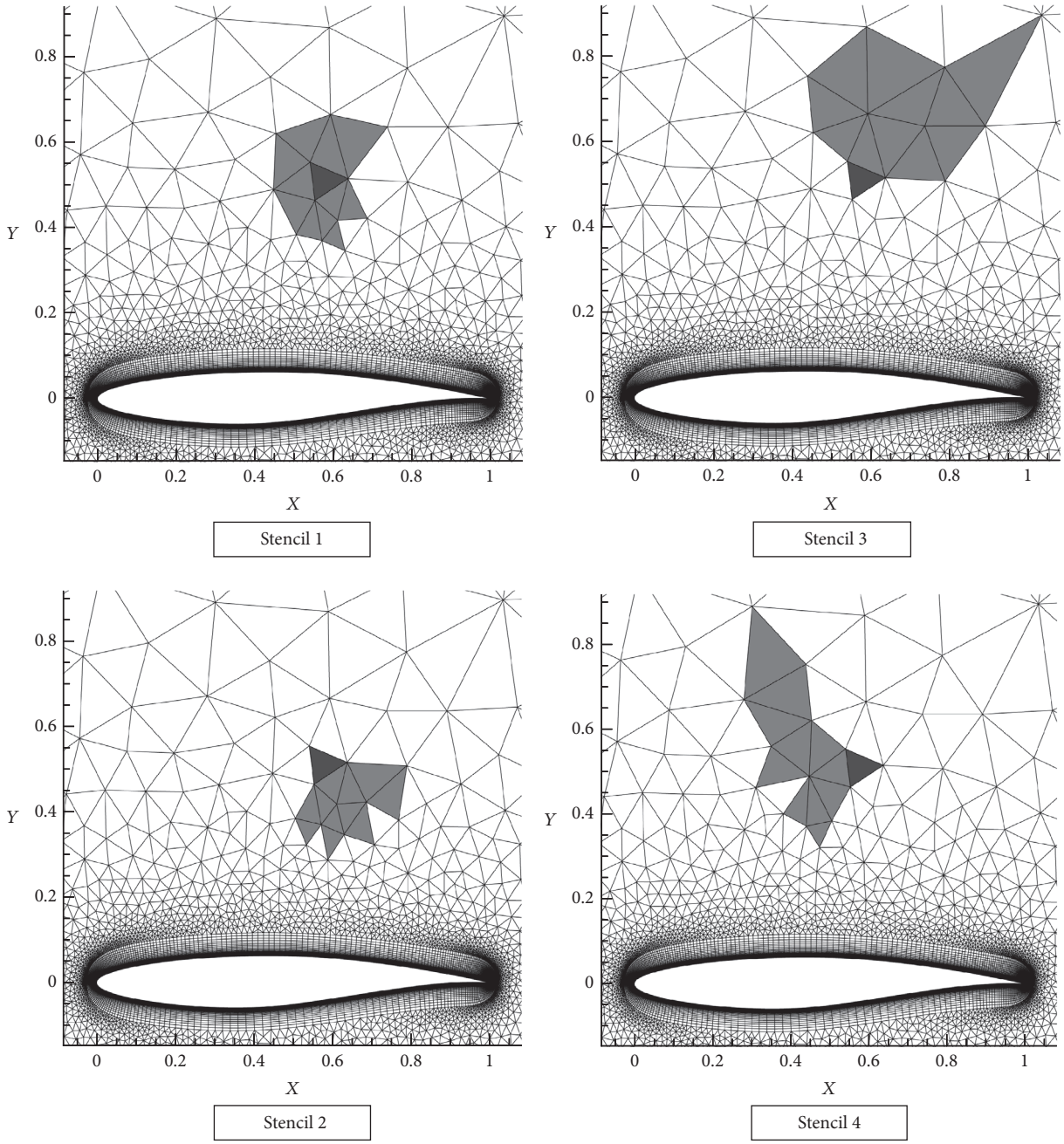

Figure 4: Central (stencil 1) and directional stencils (stencil 2, stencil 3, and stencil 4) for a triangular mesh element.

Since the most time-consuming part of the simulation is the iterative solver, the initialization and savefile output time can be neglected.

Figure 8 shows the execution time measured for MPI and Coarray Fortran version of the solver on ASTRAL (Intel compiler) and ARCHER (Cray compiler). On ASTRAL the data points correspond to $N = 8, 16$ (one-node), 32, 64, and 128 cores. On ARCHER the data points correspond to $N = 12, 24$ (one-node), 48, 96, 192, and 384 cores.

On ASTRAL with the Intel compiler, the Coarray Fortran version of the code is slower than MPI. Since the Coarray Fortran implementation of Intel is based on MPI-3 remote memory access calls, it is subject to overheads over MPI. These overheads are so big that any performance gains—by replacing the blocking send and receive commands in the MPI version to nonblocking remote access calls in the Coarray Fortran version—are wiped out. An interesting result to note is that a sudden performance degradation occurs when communication takes place among *images* in multiple nodes.

Haveraaen et al. [23] observed that, even for large messages, the Intel compiler was performing element-wise transfer when Coarray Fortran was used for communication.

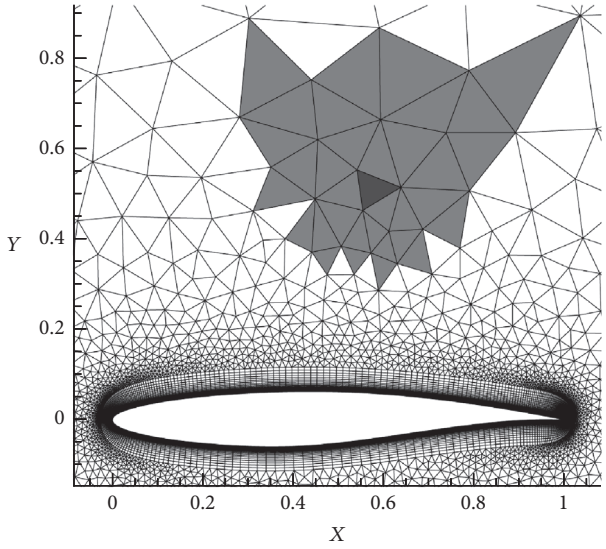

Figure 5: Combined stencil for the triangular mesh element.

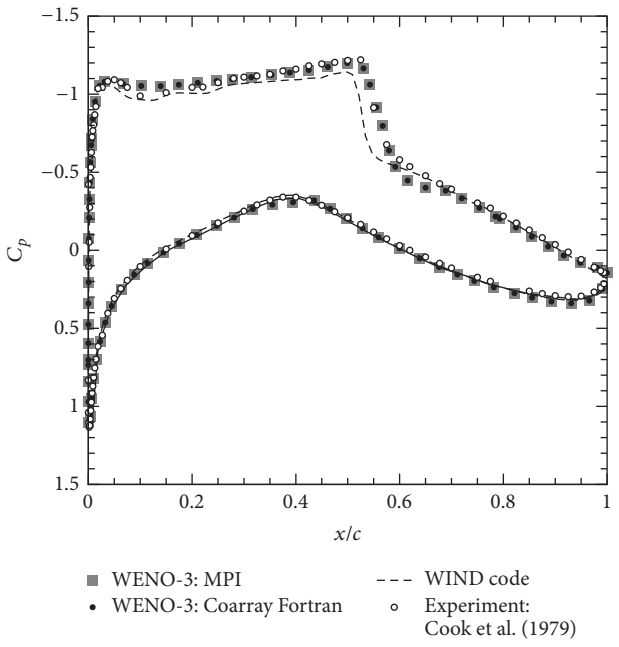

Figure 6: Pressure coefficient over the RAE 2822 airfoil surface for the code compared with experimental and CFD results from literature. Note: some points in CFD results are omitted for clarity.

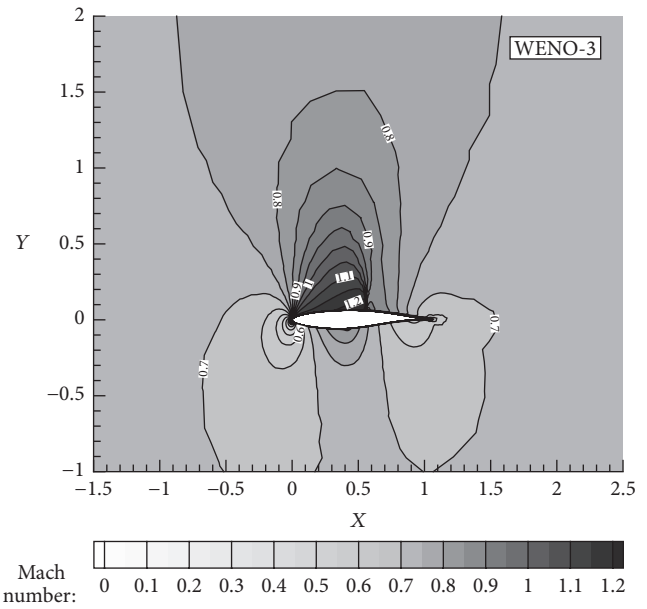

Figure 7: Mach number contour over the RAE 2822 airfoil.

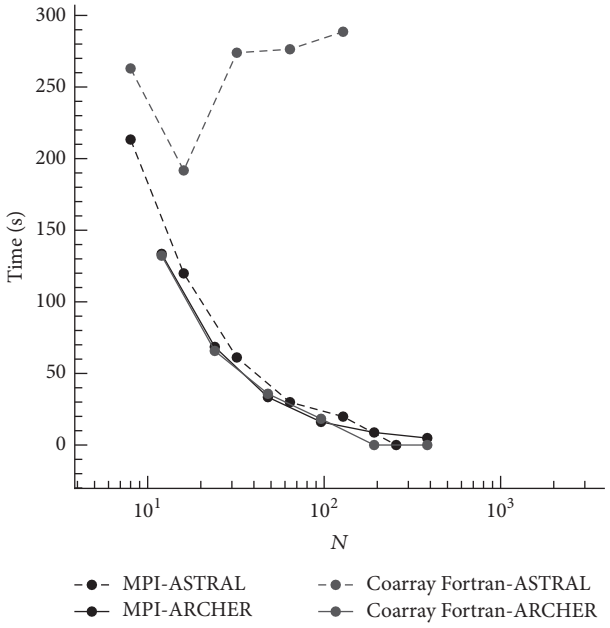

Figure 8: Performance results for the solver.

Their observation was based on the analysis of the assembly code of their program. Using Trace analysis, we also observed that the majority of communication time in each process is spent between *MPI_Win_lock* and *MPI_Win_unlock* calls, when internode communication is invoked. Thus, it could be concluded that the implementation of the Intel compiler is quite inefficient, with possible bugs that greatly reduce the performance when parallelization is performed using Coarray Fortran.

In contrast, on ARCHER with the Cray compiler, the performance of Coarray Fortran version of the code is mostly similar (till 96 cores) or in some cases (192 and 384 cores) better than the MPI version. Also, the execution time is lower on ARCHER due to the faster architecture compared to ASTRAL. For shorter messages, Coarray Fortran has lower overheads compared to MPI; this translated into the better performance when higher cores were used with the Coarray Fortran version. Also, the extraredirection due to the communication array did not adversely affect the results in comparison to the other gains.

Open-source compilers such as GCC (with OpenCoarrays) and OpenUH have been used in other benchmark studies to demonstrate the performance of their Coarray Fortran

implementation in comparison to the MPI implementations. During our study, we found that the code featured in Listing 8 is not supported by the open-source implementations.

6. Conclusions

Coarray Fortran provides a simpler and a more productive alternative to MPI for parallelization. With minimal code modifications, even codes with unstructured meshes can be parallelized with Coarray Fortran. The increased readability of the resultant code enhances the productivity. Based on the performance and the level of current development, we found the Cray compiler to be suitable for development with Coarray Fortran. The performance with Cray compiler was similar or better than MPI in the tests. With Intel compiler, significant performance degradation was observed, especially on internode communication. It may be attributed to the inefficient implementation and possible bugs. While commercial compilers support all the features of Coarrays Fortran specified in the Fortran 2008 standard, some limitations still exist with the open-source implementations, such as OpenCoarrays (GCC) and OpenUH. We are hopeful that these limitations will be resolved soon in future versions.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work used the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk). The authors also thank Dr. Panagiotis Tsoutsanis and Dr. Antonios Foivos Antoniadis from Cranfield University for their insightful discussions.

References

[1] L. Hochstein, J. Carver, F. Shull et al., "Parallel programmer productivity: a case study of novice parallel programmers," in *Proceedings of the ACM/IEEE 2005 Supercomputing Conference, SC'05*, usa, November 2005.

[2] I. Christadler, G. Erbacci, and A. D. Simpson, "Performance and productivity of new programming languages," in *Facing the Multicore - Challenge II*, vol. 7174 of *Lecture Notes in Computer Science*, pp. 24–35, Springer Berlin Heidelberg, Berlin, Germany, 2012.

[3] J. Reid, "Additional coarray features in Fortran," in *Proceedings of the 7th International Conference on PGAS Programming Models*, M. Weiland, A. Jackson, N. Johnson, and M. Fortran, Eds., 104 pages, The University of Edinburgh, Edinburgh, UK, 2013.

[4] I. Chivers and J. Sleightholme, "Compiler support for the Fortran 2003 and 2008 standards revision 20," *ACM SIGPLAN Fortran Forum*, vol. 35, no. 3, pp. 29–50, 2016.

[5] A. Fanfarillo, T. Burnus, S. Filippone, V. Cardellini, D. Nagle, and D. W. I. Rouson, "OpenCoarrays: open-source transport layers supporting coarray Fortran compilers," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14)*, Eugene, Ore, USA, October 2014.

[6] D. Eachempati, H. J. Jun, and B. Chapman, "An open-source compiler and runtime implementation for Coarray Fortran," in *Proceedings of the 4th Conference on Partitioned Global Address Space (PGAS) Programming Models, PGAS'10*, New York, NY, USA, October 2010.

[7] R. W. Numrich, J. Reid, and K. Kim, "Writing a multigrid solver using co-array fortran," in *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, vol. 1541 of *Lecture Notes in Computer Science*, pp. 390–399, Springer Berlin Heidelberg, Berlin, Germany, 1998.

[8] R. Barrett, "Co-array Fortran experiences with finite differencing methods," in *Proceedings of the The 48th Cray User Group meeting*, Italy, Lugano, Italy, 2006.

[9] M. Hasert, H. Klimach, and S. Roller, "CAF versus MPI - Applicability of Coarray Fortran to a Flow Solver," in *Recent Advances in the Message Passing Interface*, vol. 6960 of *Lecture Notes in Computer Science*, pp. 228–236, Springer Berlin Heidelberg, Berlin, Germany, 2011.

[10] A. I. Stone, J. M. Dennis, and M. M. Strout, "Evaluating Coarray Fortran with the CGPOP Miniapp," in *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS)*, 2011.

[11] A. Shterenlikht, "Fortran coarray library for 3D cellular automata microstructure simulation," in *Proceedings of the 7th International Conference on PGAS Programming Models*, 2013.

[12] D. Henty, "Performance of Fortran Coarrays on the Cray XE6," in *Proceedings of the Cray User Group*, 2012.

[13] R. W. Numrich, "A Parallel Numerical Library for Co-array Fortran," in *Proceedings of the International Conference on Parallel Processing and Applied Mathematics*, Springer Berlin Heidelberg, 2005.

[14] V. Cardellini, A. Fanfarillo, and S. Filippone, "Heterogeneous CAF-based load balancing on Intel Xeon Phi," in *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2016*, pp. 702–711, Chicago, Ill, USA, May 2016.

[15] R. W. Numrich and J. Reid, "Co-array Fortran for parallel programming," *ACM SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[16] R. W. Numrich, *Coarray Fortran*, Springer, Boston, Mich, USA, 2011.

[17] J. A. Ekaterinaris, "High-order accurate, low numerical diffusion methods for aerodynamics," *Progress in Aerospace Sciences*, vol. 41, no. 3-4, pp. 192–300, 2005.

[18] P. Tsoutsanis, V. A. Titarev, and D. Drikakis, "WENO schemes on arbitrary mixed-element unstructured meshes in three space dimensions," *Journal of Computational Physics*, vol. 230, no. 4, pp. 1585–1601, 2011.

[19] P. Tsoutsanis, A. F. Antoniadis, and D. Drikakis, "WENO schemes on arbitrary unstructured meshes for laminar, transitional and turbulent flows," *Journal of Computational Physics*, vol. 256, pp. 254–276, 2014.

[20] V. A. Titarev and E. F. Toro, "Finite-volume WENO schemes for three-dimensional conservation laws," *Journal of Computational Physics*, vol. 201, no. 1, pp. 238–260, 2004.

[21] P. H. Cook, M. C. P. Firmin, and M. A. McDonald, "Aerofoil RAE 2822: pressure distributions, and boundary layer and wake measurements," Tech. Rep., Advisory Group For Aerospace Research And Development (AGARD), 1979.

[22] J. W. Slater, J. C. Dudek, and K. E. Tatum, The NPARC alliance verification and validation archive, (2000).URL https://www.grc.nasa.gov/www/wind/valid/archive.html.

[23] M. Haveraaen, K. Morris, D. Rouson, H. Radhakrishnan, and C. Carson, "High-Performance Design Patterns for Modern Fortran," *Scientific Programming*, vol. 2015, Article ID 942059, 14 pages, 2015.

2017-06-20

MPI to Coarray Fortran: experiences with a CFD solver for unstructured meshes

Sharma, Anuj

IOS Press / Hindawi Publishing Corporation

Sharma A, Moulitsas I. (2017) MPI to Coarray Fortran: experiences with a CFD solver for unstructured meshes. Scientific Programming, Volume September 2017, Article number 3409647
https://doi.org/10.1155/2017/3409647